Highly Auditable Distributed Systems

Murat Demirbas, Sandeep Kulkarni
University at Buffalo, Michigan State

[2015-07-07 Tue]

Outline

Auditability

Auditability lets you see what is going on in the system.
Auditability helps you debug

- dependencies among events
- performance bottlenecks
- latent concurrency bugs.

Auditability enables available, scalable, and reliable distributed systems.

Auditability report card: Poor

- Logging of system messages & exceptions are OK when used on a single computer system.
- For distributed systems, where there is no shared memory & time, naive logging is insufficient.

Dapper, Zipkin, HTrace, X-Trace

These distributed system tracers perform sparse logging. They help in identifying service dependencies and long-tail latencies but fail to support nonlocal predicate detection.

Both time & causality are important for auditability

Theory of distributed systems shunned the notion of time and considered asynchronous systems whose event ordering is captured by logical clocks.

Using LC, it is not possible to query events in relation to real time. Lamport's logical clocks guarantees that
$E \underline{hb} F \implies LC.E < LC.F$

Practical distributed systems employed NTP synchronized clocks to capture time but in ad hoc undisciplined ways.

It is impractical to achieve perfect clock sync in a distributed system, so even when E causally affects F (i.e., $E \underline{hb} F$), E may have bigger timestamp than F.

Our work

By leveraging hybrid logical clocks (HLC) as a building block, we aim to bridge time & causality and design highly auditable distributed systems.

By leveraging HLC, we aim to provide

- debugging support via detectors
- availability support via correctors (to correct faults and state corruptions)

HLC implementation

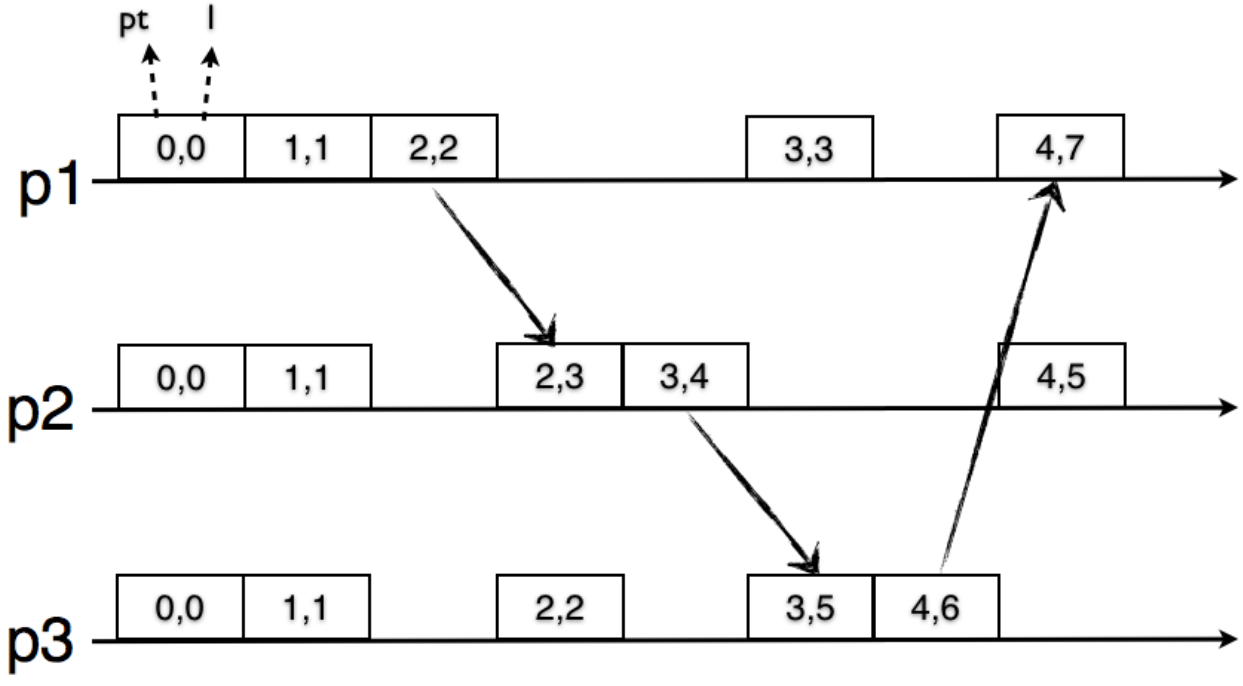
Each node j maintains timestamp of the form $\langle l.j, c.j \rangle$

- $pt.j$: physical clock of j
- $l.j$: the maximum $pt.k$ that j is aware of
- $c.j$: the length of the causal chain

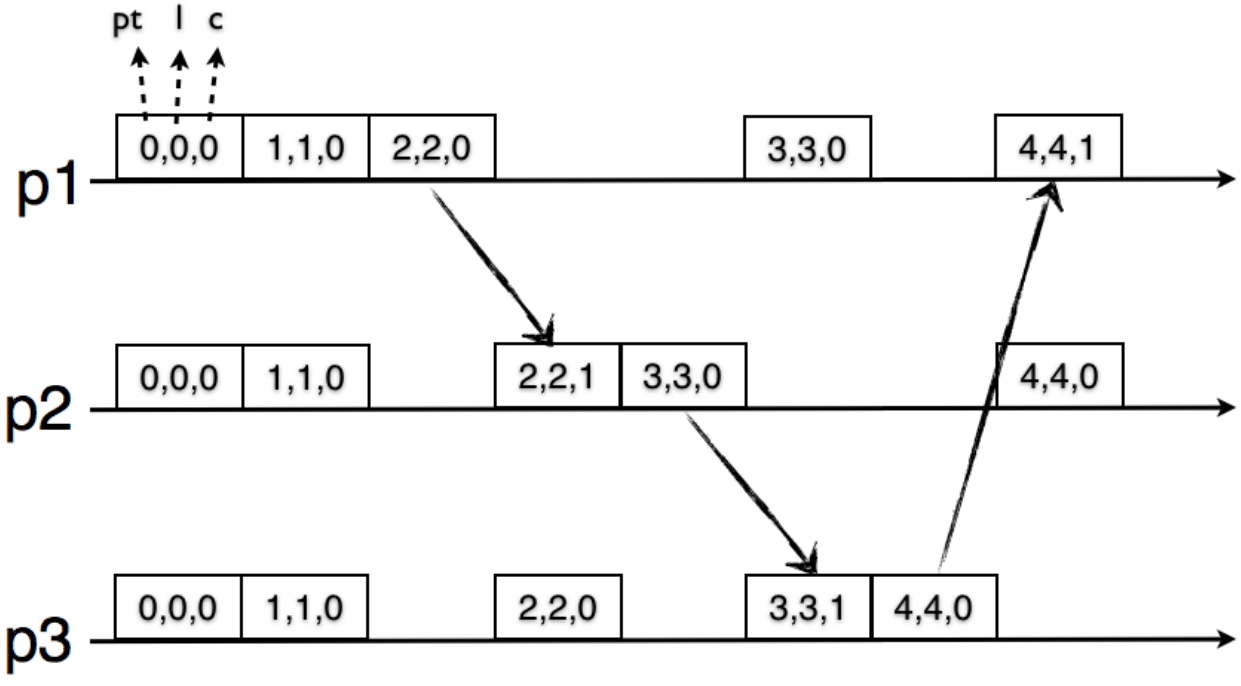
Given a maximum clock drift of ϵ, *HLC* guarantees that

- $l.j$ is in the range $[pt.j, pt.j + \epsilon]$
- $c.j$ is bounded: $c.j$ is reset to 0 when $l.j$ increases, which happens in the worst case when $pt.j$ exceeds $l.j$. Worst case bound on $c.j$ is proportional to the number of processes and ϵ. Practically, $c.j$ is less than 10 even under stress testing over AWS.

Why have $c.j$?
Otherwise $l - pt$ could grow unbounded



Why have $c.j$?
Otherwise $l - pt$ could grow unbounded



HLC algorithm

Initially $l.j := 0; c.j := 0$

Send or local event

$l'.j := l.j;$
$l.j := max(l'.j, pt.j);$
If $(l.j = l'.j)$ then $c.j := c.j + 1$
  Else $c.j := 0;$
Timestamp with $l.j, c.j$

Receive event of message m

$l'.j := l.j;$
$l.j := max(l'.j, l.m, pt.j);$
If $(l.j = l'j = l.m)$ then $c.j := max(c.j, c.m) + 1$
  Elseif $(l.j = l'.j)$ then $c.j := c.j + 1$
  Elseif $(l.j = l.m)$ then $c.j := c.m + 1$
  Else $c.j := 0$
Timestamp with $l.j, c.j$

HLC properties

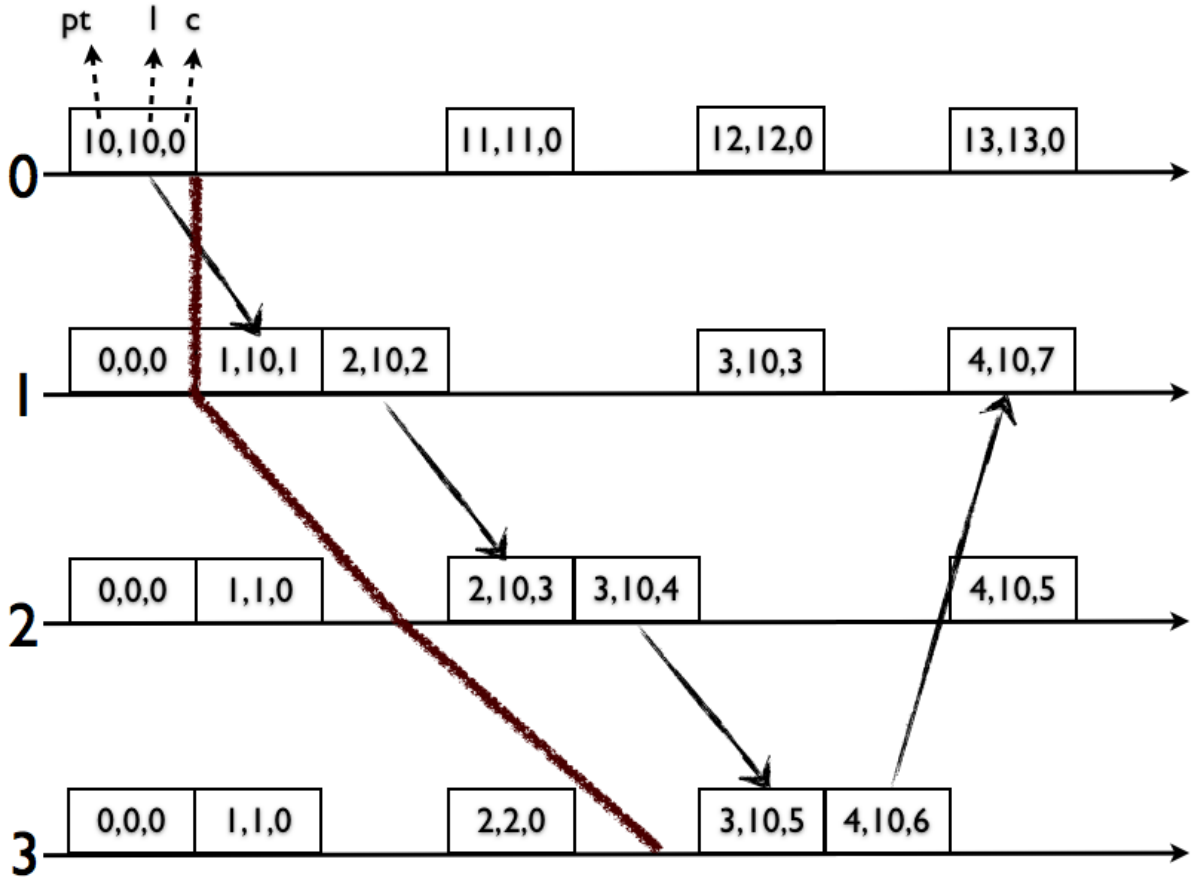
HLC has uncertainty resilience: HLC is always nonblocking and captures causality correctly even when time sync has degraded.

HLC enables highly auditable systems because HLC can provide consistent-state snapshots without needing to wait out clock sync uncertainties and requiring prior coordination.

Given that $E \underline{hb} F \implies HLC.E < HLC.F$, we have
$HLC.E = HLC.F \implies E \underline{co} F$

Snapshots can be on-demand vs. post-hoc, and online vs. offline.

HLC makes consistent snapshots easy



Finding a consistent snapshot for $pt = 10$ using HLC$=\langle 10, 0\rangle$
Here $\epsilon = 10$.

HLC versus TrueTime (TT) in Google Spanner

- The HLC-based implementation does not require waiting ϵ out, instead it records causality relations within this uncertainty window.
- HLC obviates the need for dedicated GPS or atomic clock references, and can work with NTP servers that provide an ϵ of several tens of milliseconds.
- Our HLC clocks have recently been adopted by CockroachDB, an opensource clone of Google Spanner.

Strawman1 for detection: Represent the distributed system state as a multiversion database

- Each update of a variable (i.e., version) is recorded with its associated timestamp on that node. HLC supports efficient consistent-state querying of such a distributed multiversion database.
- But recording every update of each variable is expensive.
- For stable predicates, an occasional periodic snapshot can be enough, however, for transient predicates continuous recording is needed. Big gap!

Δ stable predicates

- These are stable for at least Δ time and fill the gap between stable and transient predicates.
- A snapshot with HLC for every Δ suffices to detect these predicates. Just, record local state at every predetermined Δ epoch.
- E.g., mutual-exclusion violation (access engagement disengagement takes Δ time)
- E.g., Δ unavailability conditions that lead to Service Level Agreement violations

Predicate triggered snapshots

- The developer can provide some local predicates that act as triggers for storing of a local snapshot at a node.
- This is analogous to user-installed breakpoints for debugging a program.
- The node then forwards this trigger state to the detector and the detector performs an on-demand query for the global state with that timestamp.

Strawman1 for correction: a global reset

HLC simplifies global distributed reset

- If a reset is requested at HLC time T, the corrector can choose HLC time T'>T and require all nodes to reset to a certain state when their HLC clock reaches T'
- Since HLC refines logical clocks and respects the causality relation, it ensures that the individual resets of all nodes are *causally concurrent* even if they do not occur at the exact same instant.

However, for very large distributed systems, resetting the entire system is unacceptable (as availability suffers) and infeasible (as stragglers & corner cases cause problems).

Seamless reset of subset of nodes using HLC

1. When the corrector notices a problem at T1, it finds a subset of nodes to reset to a certain state to fix the problem.
2. The corrector sends a message to freeze these nodes at T2. The frozen nodes refrain from changing their states or communicating with any nodes until reset.
3. The corrector also takes a snapshot at T2 and determines the exact state to put the frozen nodes at T2 so that they become consistent with the rest of the system at T2.

Seamless reset of faulty nodes using HLC

By the time the frozen nodes are reset, the system time may be at T3. This is acceptable since the resulting state is still a globally consistent state as the frozen nodes refrained from communication. It is as if the frozen nodes lagged in execution.

Those reset nodes will catchup with the rest of the system. HLC is uncertainty resilient and will not lead to unsafe comparisons and updates.

Related work

- Crash-only software and restartability (Candea & Fox, HOTOS 2003)
- Eidetic distributed systems (Devecsery, Chow, Dou, Flinn, Chen, OSDI 2014)

silly poem version

auditability for distributed systems is always a crowd hit
but it remains elusive and Dapper et.al. won't cut it
the reason for this is: causality and time were studied separately
we propose a way to marry them and enable auditability

HLC is hybrid of l=logical and p=physical clocks
it uses c=counter, so l does not overtake p by many blocks
HLC is simple with c reset every time l increases by p following
yet it gives you easy consistent snapshots with an ϵ allowing
while TrueTime is blocking, HLC achieves uncertainty resilience and nonblocking

silly poem version

detectors can use HLC snapshots over multiversion databases
Δ predicates alleviates costs and recording each update eases
with predicate triggered snapshot, options for fast efficient
detection increases

for correctors, HLC simplifies global reset
or you can be precise and use HLC-based freezing of a subset

HLC improves on both logical clocks and NTP
it can be online or offline with ϵ bound on timeliness guarantee

Hybrid Vector Clocks (HVC) for more power

In the *worst case* $HVC.j$ is a vector that contains an entry for each node: $HVC.j[j]$ is the wallclock at j and $HVC.j[k]$ is the knowledge of j about the wallclock of spanserver k.

Any message sent by j includes $at.j$. Upon a message reception, j updates $HVC.j$ to reflect the max clock values that j is aware of, as in the case of VC. In contrast to VC, $HVC.j[j]$ is updated by the wallclock maintained at j.

If j does not hear (directly or transitively) from k within ϵ then $HVC.j[k]$ need not be explicitly maintained. We still infer implicitly that $HVC.j[k]$ equals $HVC.j[j] - \epsilon$, thanks to clock sync.